Working Journal: The Digital Avatar 'Ku' Project

Project Inception & Core Decisions

Objective: To create a personalized AI, a "digital avatar" named 'Ku', by finetuning a Large Language Model (LLM). The goal is for 'Ku' to emulate my communication style, persona, and possess some of my personal knowledge. The chosen base model for this endeavor is Qwen3-14B, a 14-billion parameter model.

Initial Technical Considerations: Finetuning vs. RAG

A primary decision at the outset was the core approach to personalization.

- Retrieval Augmented Generation (RAG): This method involves providing the LLM with external documents to retrieve information from at inference time. For a digital avatar of myself, this would mean creating an extensive knowledge base (personal history, opinions, style guides) for the LLM to query. While excellent for fact-based recall and incorporating up-to-date information, RAG felt less suited for deeply ingraining a *persona* or *style*. The avatar might be able to state facts about me but not necessarily communicate *like* me.
- Finetuning: This approach involves further training the base LLM on custom data, thereby adjusting its internal parameters. This seemed more promising for instilling a specific communication style, nuances, and making the persona feel more inherent to the model's responses. The aim is for 'Ku' to not just access information about me, but to *become* an extension of my digital persona.

Decided to proceed with finetuning as the primary method, believing it will better capture the desired persona and style.

Finetuning Technique: LoRA (Low-Rank Adaptation)

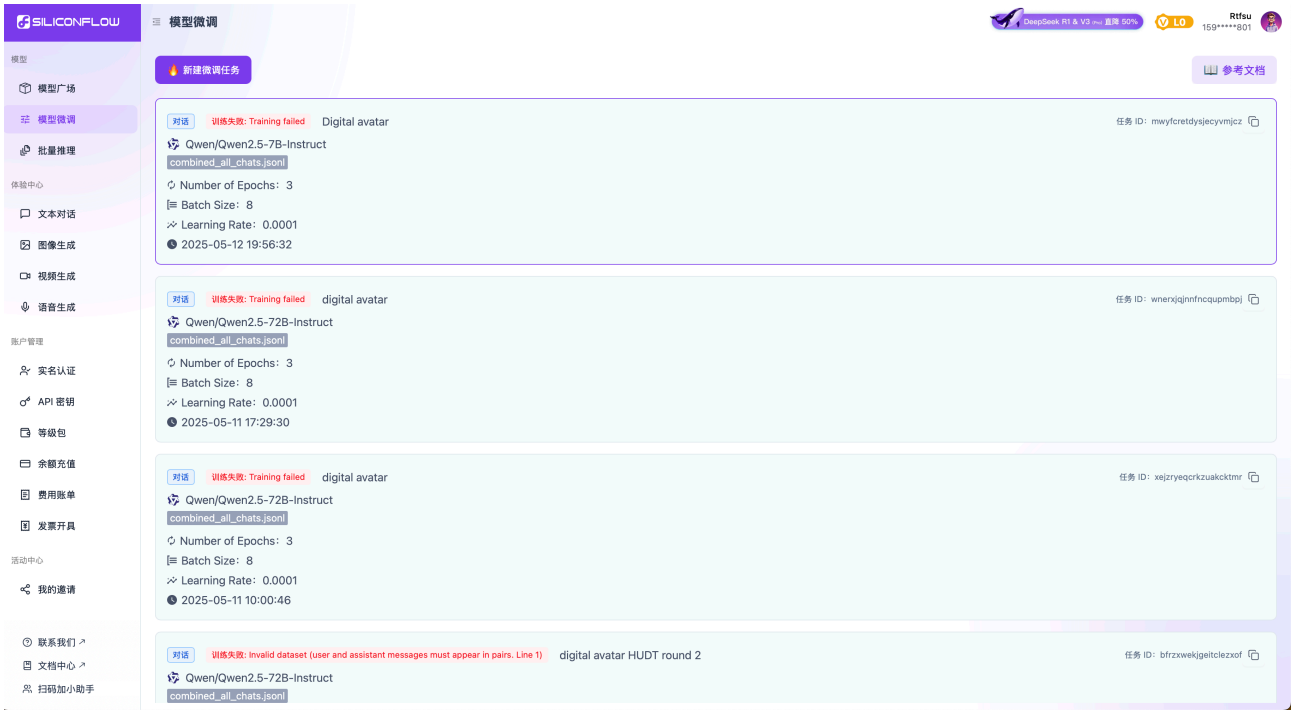
Given the scale of the Qwen3-14B model, full finetuning (retraining all parameters) would be computationally prohibitive on available resources. Therefore, LoRA was selected as the finetuning technique.

- LoRA introduces small, trainable "adapter layers" into the pre-trained LLM. Only these adapters are updated during finetuning, leaving the vast majority of the base model's parameters frozen. This significantly reduces the computational resources and memory required for training, making it feasible to finetune large models. The idea is that these adapters learn the task-specific (or in this case, persona-specific) modifications.

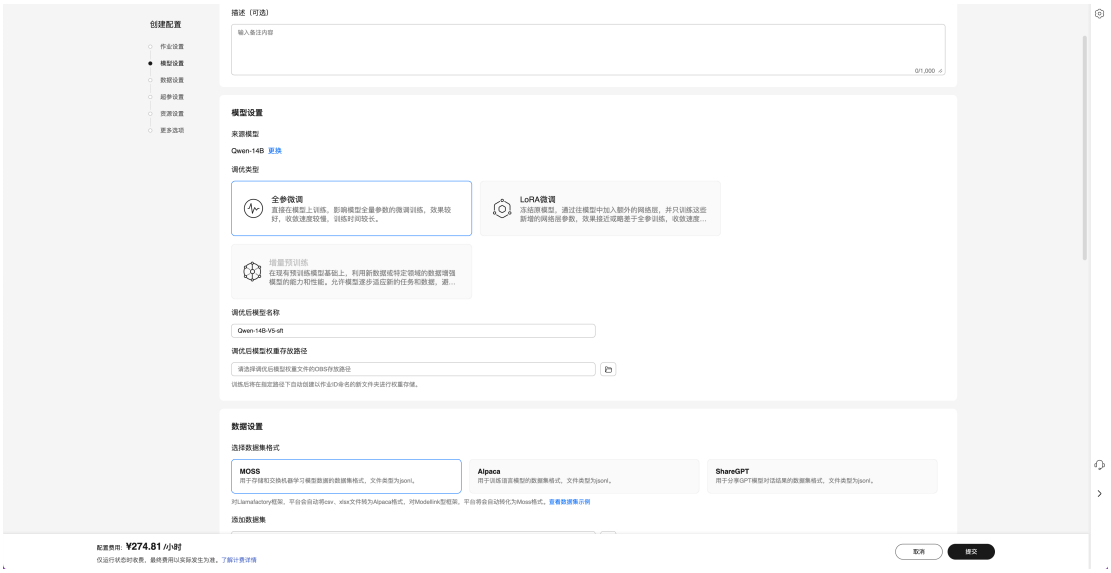
Platform Exploration for Finetuning

Before preparing the dataset, I investigated several Model-as-a-Service (MaaS) platforms to conduct the LoRA finetuning.

My initial exploration involved attempting to train on **SiliconFlow**. Unfortunately, I encountered persistent errors during the training process on this platform. A significant issue I faced was the lack of responsive technical support. The situation became critical as my access to the model on this platform ceased around May 4th. Subsequent attempts to retrain the model or contact their support yielded no response. I also observed that online discussions from other users indicated they were experiencing similar problems, which raised concerns about the platform's service stability and potentially its financial viability. This overall negative experience led to my decision to abandon SiliconFlow as a viable option for the project.

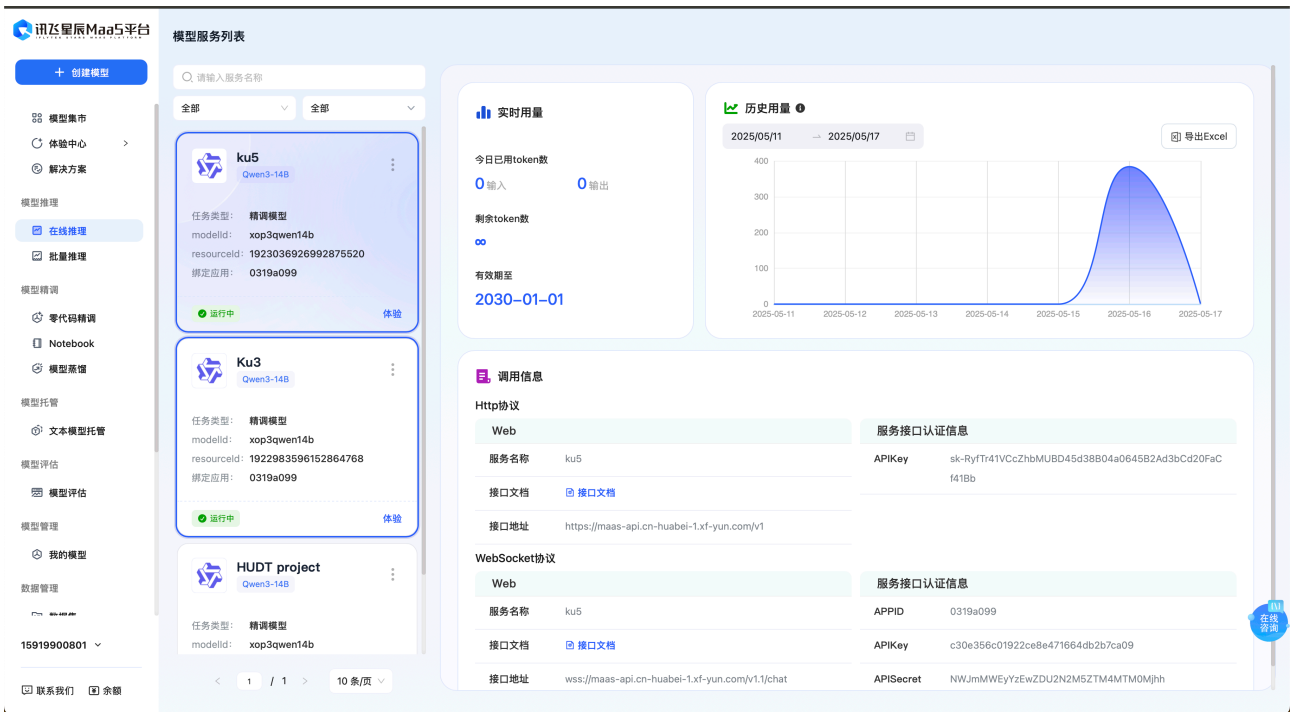


Next, I evaluated **Huawei Cloud**. This platform offered the necessary technical capabilities for training. However, a major drawback was that it required renting GPU resources at a cost of approximately 300 RMB per hour. This pricing was deemed too expensive for the project's budget.



Finally, **Xunfei Xingchen MaaS** emerged as the most suitable choice. This platform presented several key advantages. It offered free finetuning for models of the scale I was using, specifically the Qwen3-14B model. Additionally, it provided a user-friendly interface for managing training tasks. Crucially, Xunfei Xingchen MaaS also offered robust API support, which I knew would be essential for later interacting with the finetuned model programmatically and for building the chat interface.

Based on this evaluation, the **Decision** was made to proceed with Xunfei Xingchen MaaS for finetuning 'Ku'.

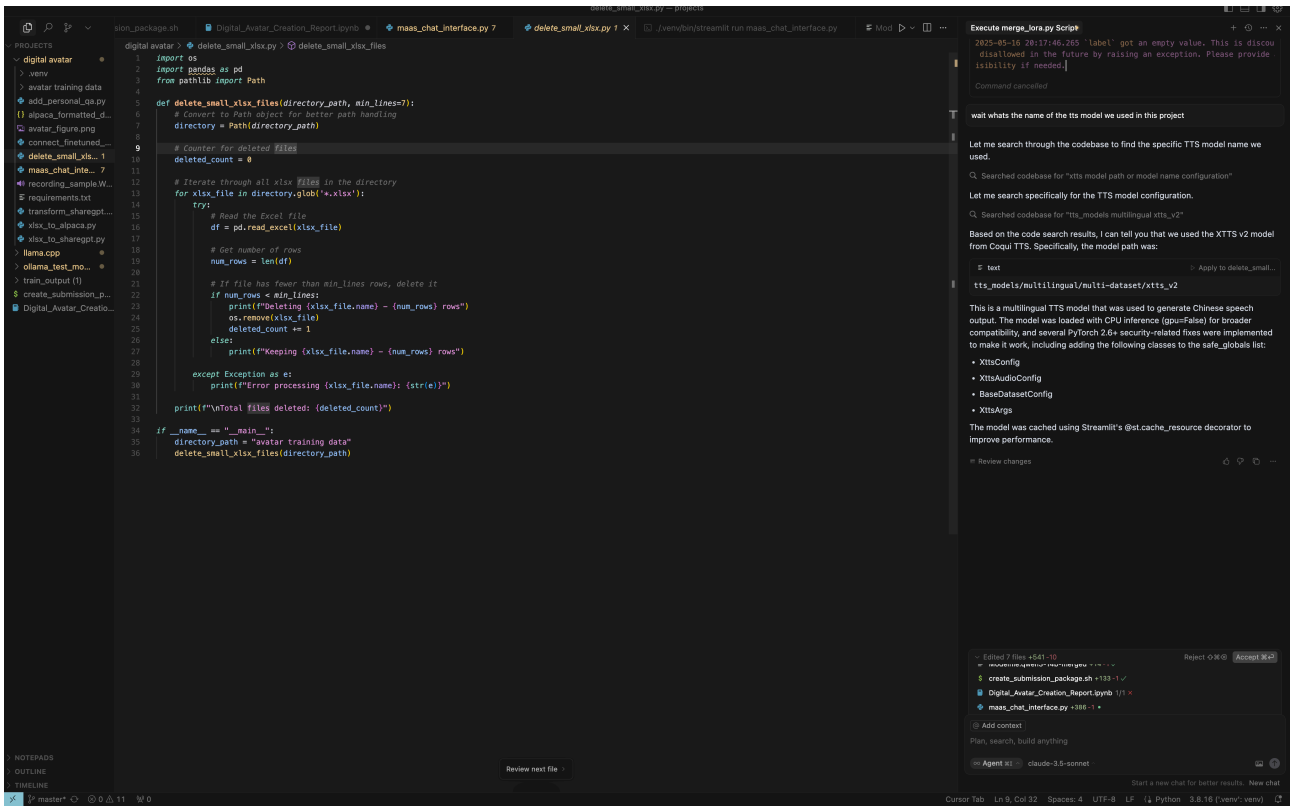


Data Collection, Cleaning, and Preparation for 'Ku'

With the platform selected, the focus shifted to preparing a high-quality dataset to train 'Ku'.

- **Data Source:** The primary data consisted of my personal chat logs, primarily exported from WeChat. These were in numerous Excel (.xlsx) files, each representing a conversation with a different individual, stored in a directory named `avatar training data`.

- **Data Cleaning (`delete_small_xlsx.py`):**



Data Conversion to Training Formats:

- The MaaS platform required data in a specific JSON format. My process involved an initial attempt with the ShareGPT format, followed by a pivot to the Alpaca format due to platform compatibility issues.

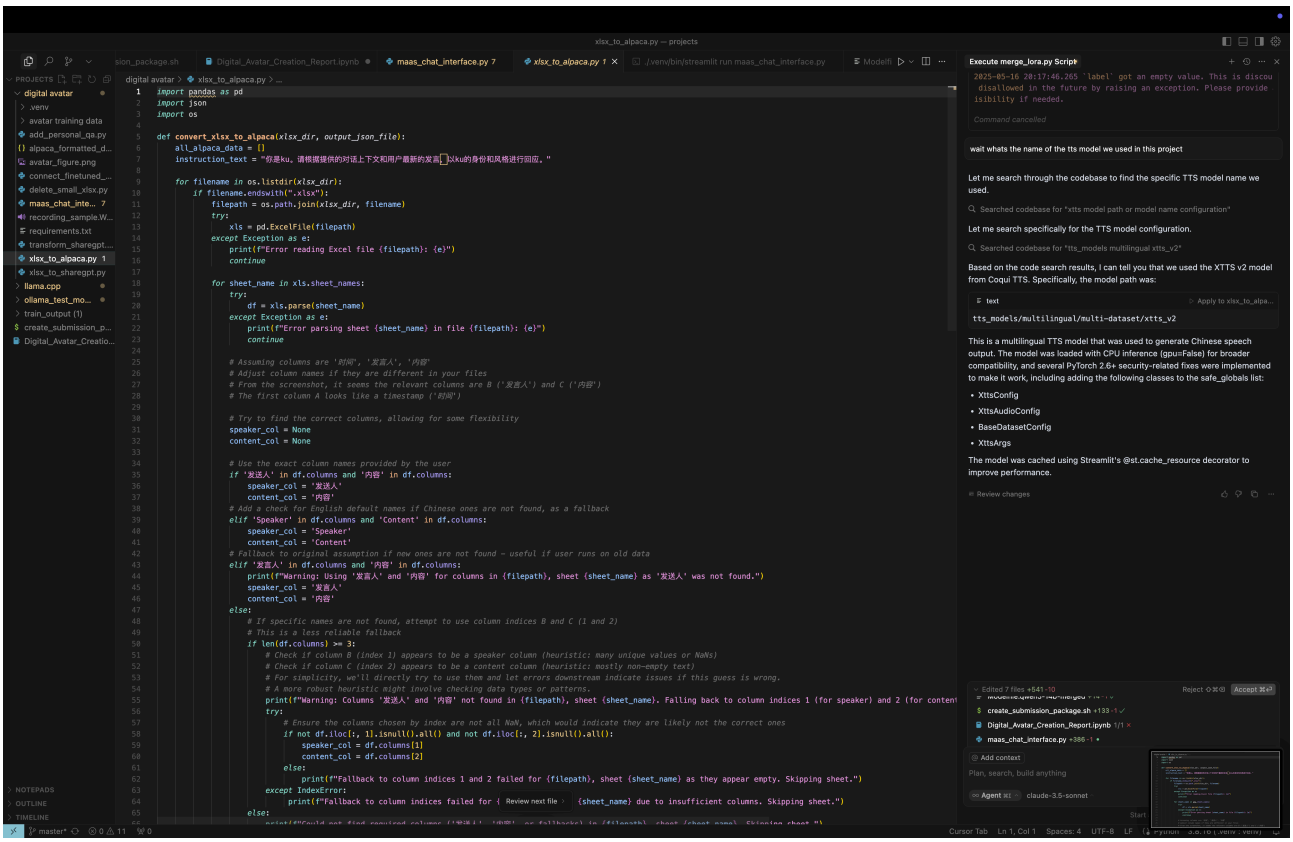
- **Initial Preparation in ShareGPT Format (`xlsx_to_sharegpt.py`):**

I first developed the `xlsx_to_sharegpt.py` script to convert the cleaned Excel chat logs into the ShareGPT JSON structure.

The script's `convert_xlsx_to_sharegpt(input_dir, output_file)` function processes each Excel file from the `input_dir`. It extracts messages by referencing specific column names, for example, `sender_col = '发送人'` and `message_col = '内容'`.

It structured each conversation with an initial system prompt, as seen in the code: `messages.append({"from": "system", "value": "你是ku。请根据对话内容自然回应。"})`.

Subsequent messages were then added by mapping my messages (identified if `sender.lower() == "ku"` or if `sender_val` was NaN, implying it was me) to `{"from": "ku", "value": message_text}` and the other person's messages to `{"from": "user", "value": message_text}`. However, upon testing the output file (`all_conversations_sharegpt.json`) on the Xunfei platform, I encountered glitches and errors that cannot be identified, I can only try to use another format called Alpaca.



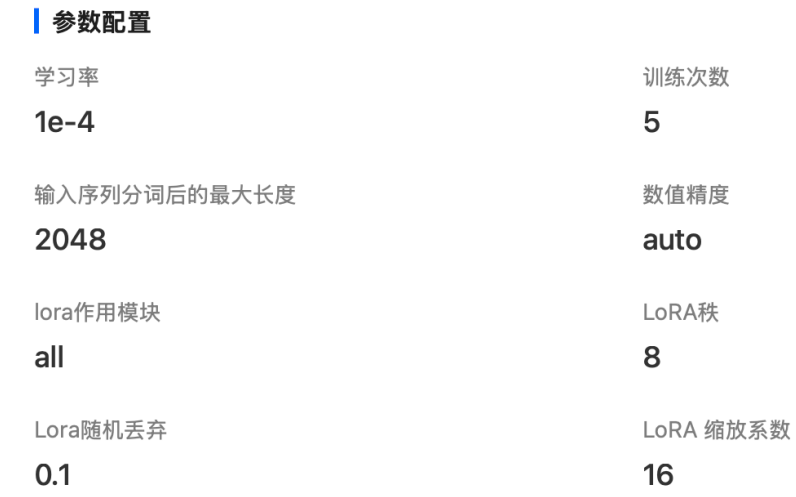
Pivoting to Alpaca Format (`xlsx_to_alpaca.py`):

- Due to the issues with ShareGPT, I then focused on the Alpaca format, developing the `xlsx_to_alpaca.py` script.
- The `convert_xlsx_to_alpaca(xlsx_dir, output_json_file)` function in this script also iterates through the Excel files. It processes multi-turn conversations and structures them into Alpaca's instruction-input-output format.
- A key instruction, defined in the script as `instruction_text = "你是ku。请根据提供的对话上下文和用户最新的发言，以ku的身份和风格进行回应。"`, was used for each entry. This instruction guides the model during finetuning.
- The script's logic for handling turns is crucial: it accumulates messages from the 'user' (the other person in the chat) into a `user_input_buffer`. When a message from 'ku' (me,

identified by `pd.notna(speaker) and str(speaker).strip()` being false) is encountered, it forms an Alpaca item:

Finetuning 'Ku' on Xunfei Xingchen MaaS

- The prepared `alpaca_formatted_data.json` file was uploaded to the Xunfei platform.
- A LoRA finetuning job was configured and initiated, targeting the Qwen3-14B base model.
- **Parameter Configuration (参数配置):** The finetuning process on the Xunfei platform involved setting several key hyperparameters. These were chosen based on common practices for LoRA finetuning and the specifics of the platform's interface, aiming for a balance between effective learning and resource efficiency:



  - **学习率 (Learning Rate): `1e-4` (0.0001)**
    - This is a relatively standard learning rate for finetuning LLMs with LoRA. A smaller learning rate helps the model make gradual adjustments, preventing it from diverging or "forgetting" what the base model already knows.
  - **训练次数 (Training Epochs): `5`**

        - An epoch represents one full pass through the entire training dataset. Training for 5 epochs was chosen as a starting point to allow the model sufficient exposure to the data to learn the 'Ku' persona without overfitting, which can happen with too many epochs, especially on smaller datasets.

    - **输入序列分词后的最大长度 (Max Sequence Length after Tokenization): `2048`**

        - This parameter defines the maximum number of tokens (words or sub-words) that the model will consider in a single input sequence. A length of 2048 allows for reasonably long conversational contexts to be processed, which is important for maintaining coherence in dialogue. This value is often a good balance for Qwen-series models.

    - **数值精度 (Numerical Precision): `auto`**

        - Setting this to `auto` allows the platform to choose the optimal numerical precision (e.g., float16, bfloat16, or float32) based on the available hardware and model requirements. This often leads to a good trade-off between training speed, memory usage, and model performance.

    - **LoRA作用模块 (LoRA Target Modules): `all`**

        - This setting indicates that LoRA adapters were applied to all available linear layers (or a predefined comprehensive set of layers like attention and feed-forward network layers) in the Qwen3-14B model that are typically targeted by LoRA. Applying LoRA to 'all' such modules provides more capacity for the model to learn the new persona. The `adapter_config.json` later confirmed the specific `target_modules` as `["o_proj",`

`"q_proj", "up_proj", "down_proj", "gate_proj", "k_proj", "v_proj"]`.

- **LoRA秩 (LoRA Rank / r): `8`**
  - The rank (`r`) in LoRA determines the dimensionality of the trainable adapter matrices. A smaller rank means fewer trainable parameters. A rank of `8` is a common and effective choice, offering a good balance between model adaptability and parameter efficiency. It allows the model to learn effectively without drastically increasing its size. This was also reflected in the `adapter_config.json` (`"r": 8`).
- **Lora随机丢弃 (LoRA Dropout): `0.1`**
  - Dropout is a regularization technique used to prevent overfitting. A value of `0.1` means that 10% of the LoRA adapter activations are randomly set to zero during training. This helps the model generalize better to unseen data. This value is also present in the `adapter_config.json` (`"lora_dropout": 0.1`).
- **LoRA 缩放系数 (LoRA Alpha / Scaling Factor): `16`**
  - LoRA alpha is a scaling factor for the LoRA weights. It's often set in relation to the rank (`r`). A common practice is to set alpha to be twice the rank (alpha = 2 * r), or simply a value like 16 or 32. Here, alpha (`16`) is twice the rank (`8`), which is a standard configuration. This scaling helps balance the influence of the LoRA adapter with the pre-trained model weights. This was

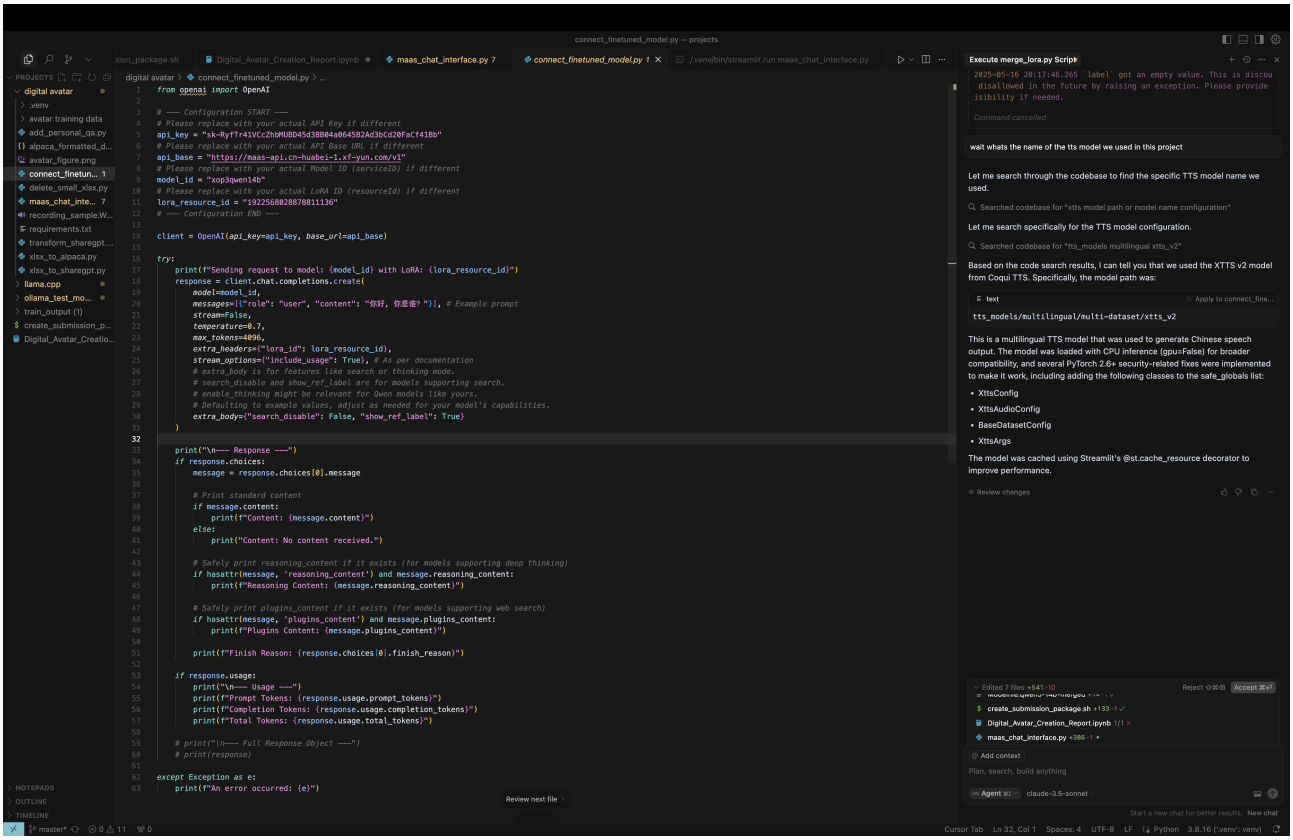
confirmed by `"lora_alpha": 16` in the `adapter_config.json`.

- The platform handled the training process based on these configurations.
- Upon completion, the platform provided a `lora_resource_id` for the successfully finetuned 'Ku' adapter. This ID is crucial for later invoking the finetuned model via API.

Ivnteraction, Interface Development, and Voice Integration

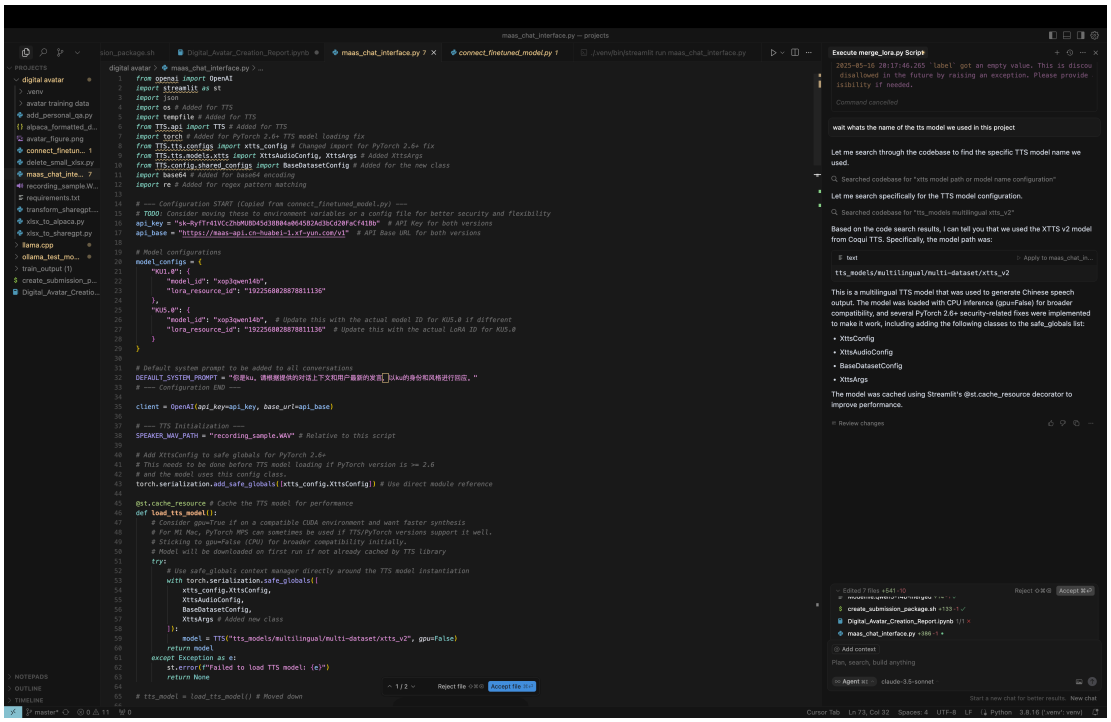
With the 'Ku' LoRA adapter trained, the next phase was to interact with it and build a user-friendly interface.

- **API Connection (`connect_finetuned_model.py`):**



Streamlit Chat Interface (`maas_chat_interface.py`):

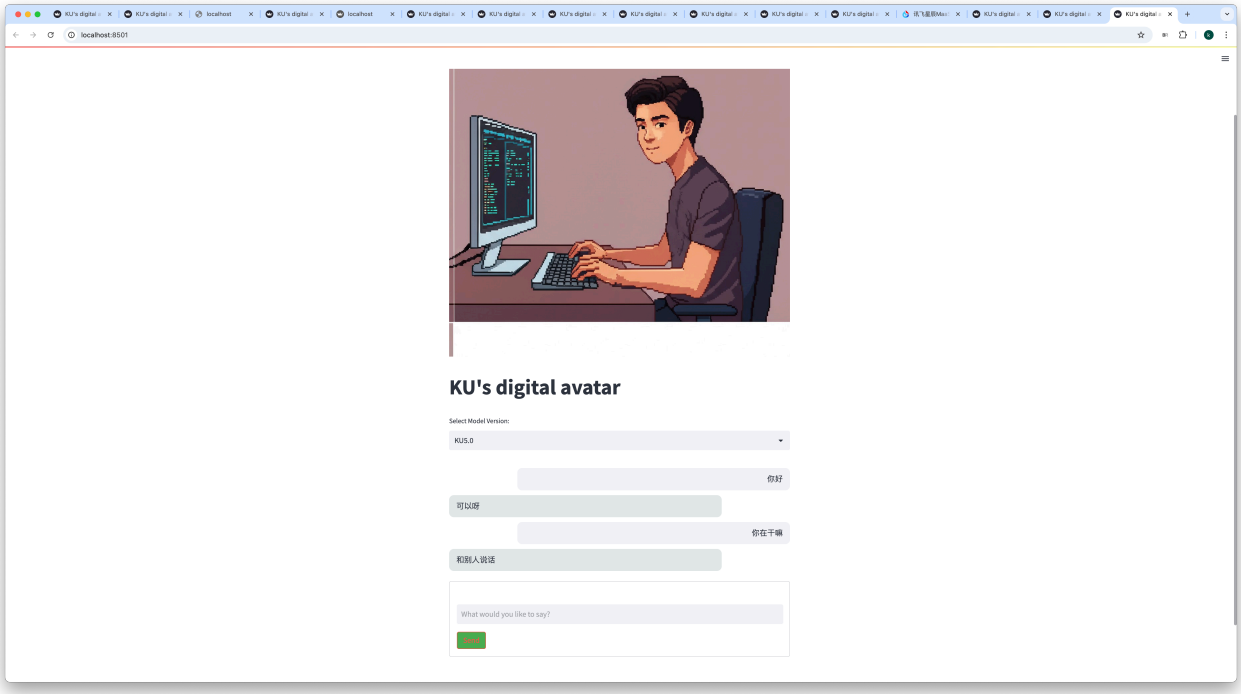
- To create a more engaging interaction experience, I built a web-based chat interface using Streamlit.



- **Core Functionality:**
  - The application, `maas_chat_interface.py`, provides a chat window.
  - It's titled `"KU's digital avatar"` and displays a custom avatar image (`st.image("avatar_figure.png")`).
  - It maintains conversation history using `st.session_state.messages`.
  - User input is captured via `st.text_input` within an `st.form`.
  - The `get_model_response(current_conversation_history)` function calls the MaaS API, ensuring the `DEFAULT_SYSTEM_PROMPT =`

"你是ku。请根据提供的对话上下文和用户最新的发言，以ku的身份和风格进行回应。" is added and the correct `lora_resource_id` is used based on the selected model version.

- **Model Versioning:** An important feature is the ability to switch between different 'Ku' LoRA adapter versions (e.g., "KU1.0", "KU5.0"). This is managed via `st.selectbox` and a `model_configs` dictionary holding the respective `model_id` and `lora_resource_id` for each version. The chat history (`st.session_state.messages`) clears upon changing models.
- **Emoji Handling:** The interface processes 'Ku's' text responses to replace placeholders like "[微笑]" with their corresponding emojis (e.g., `assistant_response = assistant_response.replace("[微笑]", "😊")`), enhancing naturalness.



Voice Integration (TTS):

- To make the avatar more immersive, Text-to-Speech (TTS) functionality was integrated into the Streamlit app.I used the `TTS` library from Coqui-AI (`from TTS.api import TTS`), specifically `tts_models/multilingual/multi-dataset/xtts_v2`, which supports voice cloning. This is initialized in the `@st.cache_resource def load_tts_model():` function within `maas_chat_interface.py`. A voice sample, defined by SPEAKER_WAV_PATH = "recording_sample.WAV", was provided to the TTS model.

- **Process:** When 'Ku' generates a text response, the Streamlit app sends this text to the loaded TTS model. The function call is `tts_model.tts_to_file(text=assistant_response, speaker_wav=speaker_arg, language=tts_language, file_path=output_audio_path)`.

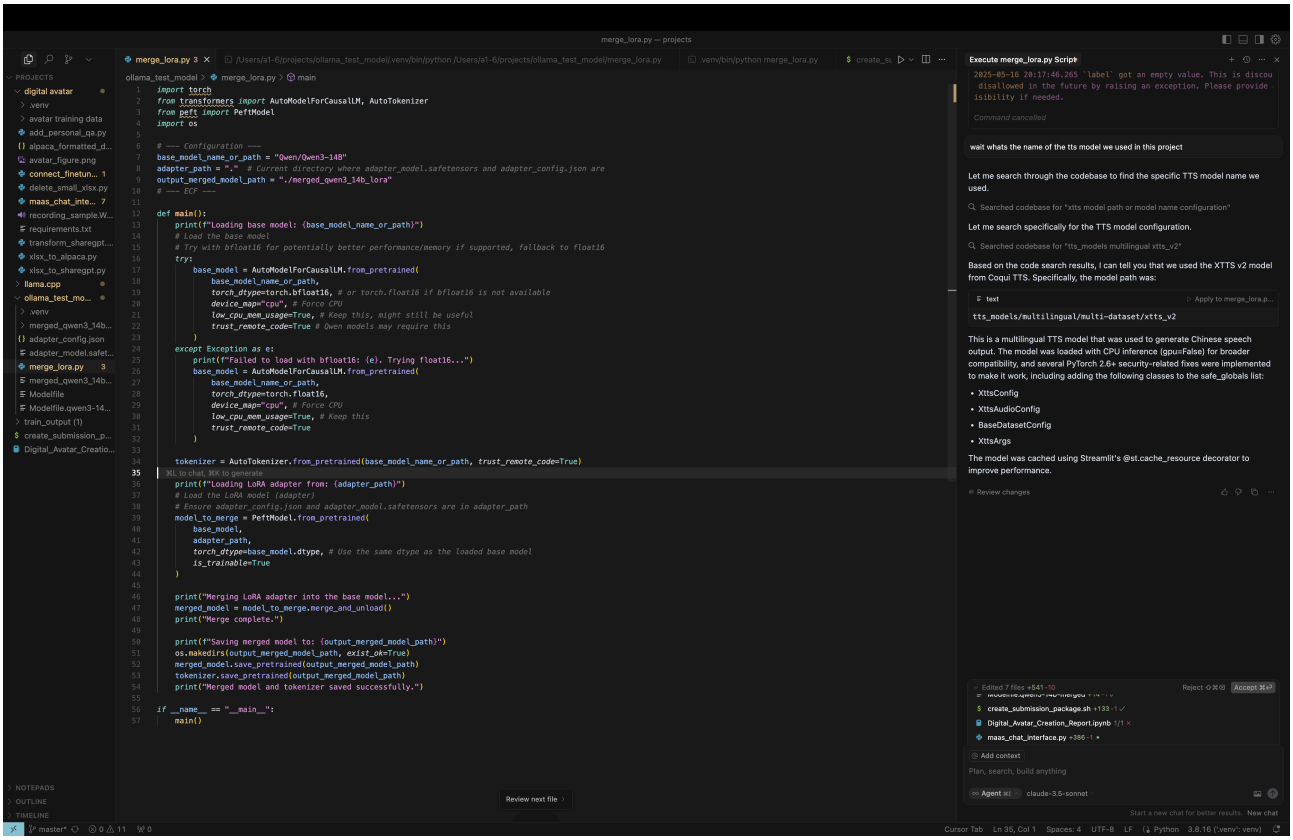
- The synthesized audio is then base 64 encoded (`audio_base64 = base64.b64encode(audio_bytes).decode()`) and played automatically in the browser using an HTML `<audio autoplay class="hidden-audio">` tag. This allows 'Ku' to not just type but also *speak* its responses in a voice cloned from mine.

- Logic is included (`if tts_model and assistant_response and not contains_emoji and not is_only_punctuation(assistant_response):`) to ensure TTS is only attempted on suitable text (e.g., not on responses that are only emojis or placeholders).

In short, this is how I developed fully interactive Streamlit web application allowing real-time, spoken conversations with different versions of the finetuned 'Ku' digital avatar.

Local Deployment Exploration & Challenges

While the cloud-based MaaS solution was effective, I also explored the feasibility of running 'Ku' locally. This involved attempting to merge the downloaded LoRA adapter with the base model for local inference.



- **Process & Tools:**
    - **Downloaded LoRA Adapter:** The trained LoRA adapter files (`adapter_model.safetensors` and `adapter_config.json`) were downloaded from the Xunfei platform. The `adapter_config.json` specifies crucial LoRA parameters such as:
    - **Planned Next Step (GGUF Conversion):** The intention after successfully merging was to convert this merged model into the GGUF format, which would then allow it to be run locally using a framework like `llama.cpp`. This step was contingent on a successful and performant merge.

- **Hardware Limitations:**
    - Attempting to run the merge script and subsequently the 14-billion parameter model (even with just the LoRA adapter active before a full merge for inference, or the merged model itself) on my MacBook Pro proved challenging.
    - The primary bottleneck was insufficient VRAM (显存). The model's memory requirements, especially during the loading and merging process which was forced to CPU (`device_map="cpu"`) due to initial GPU memory errors, far exceeded what my laptop could provide efficiently While the merge script itself could eventually complete on CPU (albeit slowly), the subsequent inference speed for such a large model on CPU-only or with insufficient VRAM was extremely low. Testing indicated text generation at a rate of approximately one word every two minutes, making it practically unusable for interactive conversation

This experiment clearly demonstrated that while LoRA makes finetuning more accessible, running and even merging large models locally still demands substantial hardware resources, particularly GPU VRAM for efficient operation.With access to more powerful lab equipment,local deployment, including the merge process and subsequent inference, would be entirely feasible and much more performant

Challenges

Several challenges were encountered throughout this project. Platform reliability and support proved to be a significant concern, particularly with the issues experienced on SiliconFlow, such as unresponsiveness and potential service discontinuation, which highlighted the risks of relying on third-party platforms. Ensuring data was in the correct format for the chosen MaaS platform also required iteration and troubleshooting, especially when pivoting from the initially planned ShareGPT format to the Alpaca format due to compatibility issues. Furthermore, computational resources for local deployment were a major hurdle; the significant VRAM limitations on my personal MacBook made local inference and the efficient merging of the 14B model impractical. As with many Python projects, managing dependencies for various libraries—including OpenAI, Streamlit, TTS, pandas, transformers, and peft—required careful environment setup, although the `requirements.txt` file was helpful in managing some of these. Lastly, the cost of commercial cloud GPUs, such as those on Huawei Cloud, can be a prohibitive barrier for individual projects with limited budgets.

Overall, this project provided a comprehensive, hands-on experience in the end-to-end process of creating a personalized AI. It effectively demonstrated the capabilities of current MaaS platforms for making finetuning accessible, while also highlighting the ongoing challenges and important considerations related to local deployment. It represents a valuable step into the practical application of LLMs for developing personalized digital experiences.

Future Improvements & Next Steps

Looking ahead, several avenues for improvement and further development have been identified. To enhance 'Ku's' knowledge and style, I plan to continuously refine the avatar by incorporating more diverse and extensive training data, such as varied chat logs or other forms of my personal writings, which should broaden its knowledge base and make its conversational style even more nuanced.

Further experimentation with different LoRA versions (KU1.0, KU5.0, and potentially new iterations) using different training strategies will also be crucial for observing how the persona can evolve.

Improving the quality of the voice cloning is another area of focus. This could involve exploring more advanced Text-to-Speech models (MinMax) or providing more extensive and higher-quality voice samples to train the voice clone, aiming to increase naturalness and reduce any robotic artifacts in the speech output.

Another significant future step is to pursue a more robust local deployment of 'Ku'. This would involve testing the model on machines with significantly more GPU VRAM to validate performance and confirm the feasibility of a truly private and offline digital avatar, including the efficient execution of the LoRA merge script.

Finally, given the platform reliability issues encountered, future projects relying on MaaS will necessitate more thorough due diligence regarding platform stability, long-term service guarantees, and the responsiveness of technical support.